Université Bordeaux 1
Master Informatique
Année 2010-2011

Plate-forme mobile d'administration d'objets communicants

Mémoire de Projet de Fin d'Études

Sébastien Barascou
<sebastien.barascou@etu.u-bordeaux1.fr>
Geoffrey Boyer
<geoffrey.boyer@etu.u-bordeaux1.fr>
Loïc Fournier
<loic.fournier@etu.u-bordeaux1.fr>
Yann Hauquin
<yann.hauquin@etu.u-bordeaux1.fr>

Tuteurs : Jérémie Albert - Serge Chaumette

Résumé

Ce mémoire présente le projet de fin d'études "Plate-forme mobile d'administration d'objets communicants" de la spécialité Systèmes & Réseaux du Master II en Informatique de l'Université de Bordeaux I.

Le projet a eu un double but, didactique et technique. En effet, sa problématique était de devoir découvrir les plates-formes électroniques modulaires "Arduino", de tester leurs possibilités, et d'arriver à pouvoir communiquer avec elles et les administrer à distance depuis un terminal mobile Android.

Ce document est lui-même à la fois didactique et technique. Dans un premier lieu, nous y présentons le matériel et ses spécificités, et dans un second lieu les différentes étapes de notre projet. Nous expliquons les divers problèmes que nous avons rencontré, les solutions et réponses variées que nous avons compilé, tout ceci dans le but d'en faire une source d'information relative au développement Arduino et Android.

Table des matières

I Introduction . . . 2
1 Sujet officiel . . . 2
2 Compréhension du sujet . . . 2
3 Matériel à disposition . . . 3
4 Étude préalable de la technologie Arduino . . . 3
5 Environnement de développement Arduino . . . 5
6 Environnement de développement Android . . . 6
II Présentation des *shields* Arduino . . . 9
1 *Shield* XBee . . . 9
2 *Shield* Wi-Fi . . . 11
3 *Shield* GPS . . . 14
4 *Shield* GSM/GPRS . . . 16
III Élaboration du projet . . . 19
1 Création & Configuration de réseaux Wi-Fi Arduino . . . 19
2 Protocole de communication . . . 20
3 Module tracker GPS . . . 20
4 Module d'information de luminosité . . . 21
5 Module d'information de température . . . 22
6 Module de détection d'ouverture de porte . . . 23
7 AWACS . . . 25
IV Conclusion . . . 28

Bibliographie **29**

I Introduction

1 Sujet officiel

Encadrants : Jérémie Albert, Serge Chaumette
Mots-clés : Android, Arduino, tablettes, réseaux mobiles ad hoc
Matériels mis à disposition : Tablettes Android, processeurs et *shields* Arduino
Équipe de recherche concernée : Thème Muse de l'équipe LSR du LaBRI

La prolifération des systèmes embarqués et l'apparition de nouvelles technologies autour des réseaux sans fil ont permis la conception d'applications novatrices ciblant l'*Internet des Choses*. Le réseau (ad hoc) sous-jacent est uniquement formé par les liens mis en place deux à deux par les objets communicants. Le présent projet vise à fournir une plate-forme de gestion de ces matériels/réseaux. Il a deux objectifs : technique et didactique.

Le travail demandé a pour objet de permettre la prise en main de ces équipements et de fournir une plate-forme d'administration permettant de les gérer.

Il faudra en particulier réaliser les étapes suivantes :
- Installation et mise en oeuvre sur des tests des plates-formes Arduino et tablettes Android
- Réalisation d'objets communicants à base de technologie Arduino
- Mise en place de communications Arduino/tablette Android
- Conception et mise en oeuvre d'une plate-forme d'administration d'équipements Arduino sur tablette Android

2 Compréhension du sujet

De nouvelles technologies et manières de penser doivent être créées pour exploiter au maximum le potentiel des nouveaux genres de terminaux mobiles (smartphones et PDAs), qui sont à la jonction au niveau de la puissance et des fonctionnalités entre l'ordinateur et le téléphone mobile.

Par exemple, là où auparavant était utilisé le modèle d'échanges dit "par infrastructure" requérant un serveur centralisé auquel devaient se connecter les clients, le modèle de réseau "ad hoc" a été créé, du latin pour dire "formé dans un but précis". Ce modèle permet à deux terminaux mobiles de se connecter l'un à l'autre, en évitant le modèle client/serveur et sans avoir à passer par un serveur central.

Ce modèle vient évidemment avec ses avantages, comme son efficacité (les données passent directement à leur destinataire plutôt que de parcourir un trajet inutile), mais aussi ses défauts liés à ses spécificités.

Ces défauts sont donc à prendre en compte lors du développement et ainsi sont nécessaires :
- La gestion des contraintes matérielles des terminaux mobiles (bande-passante et puissance de calcul limitée, consommation d'énergie à limiter)
- La gestion de la mobilité des clients (détection de nouveaux nœuds, connexions au réseau, perte de routes entre deux clients, disparition d'un nœud du réseau)

- La gestion du routage (pas d'adresses "fixes", transmission des données nœud à nœud pour arriver au destinataire final)
- La gestion de la latence et des interférences / éventuelles pertes de données induites par l'environnement
- La gestion des limites de visibilité entre clients (nœuds cachés, duplicatas d'adresses)
- La gestion de la sécurité des données

Le projet de notre équipe vise à fournir une plate-forme d'administration capable de gérer ces problèmes quant à la communication entre entités mobiles, et nous travaillerons plus particulièrement sur des tablettes Android et des processeurs/*shields* (extensions) Arduino.

3 Matériel à disposition

Nous avons reçus dans le cadre du projet le matériel suivant :

- 2 tablettes Huawei Ideo S7 (système d'exploitation Android)
- 3 Starter Kits Arduino (carte *UNO*)
- 1 Sparkfun Inventor Kit Arduino (carte *DUEMILANOVE*)
- 4 modules Wi-Fi (Wifly)
- 4 modules XBee / 4 Puces XBee (séparées, à combiner)
- 2 modules GSM (avec carte, adaptateur antenne et antenne séparée)
- 2 modules GPS
- 4 modules écrans
- 2 blocs d'alimentation secteur (sortie de 6V à 12V)
- 2 blocs d'alimentation externe avec piles fournies (sortie 9V)

4 Étude préalable de la technologie Arduino

Arduino est une plate-forme technologique faite pour qu'un terminal puisse détecter d'autres entités et communiquer de manière accessible avec elles. Il s'agit aussi bien d'un environnement de développement et de programmation qu'une technologie matérielle, c'est à dire une carte électronique. Les spécifications aussi bien matérielles que logicielles d'Arduino sont open source.

Le logiciel de programmation des modules officiels Arduino a été codé en Java, et sert d'éditeur de code aussi bien que de compilateur. Il peut également transférer les mises à jour du firmware et les programmes vers la carte Arduino.

Le langage de programmation Arduino est une variante allégée du C/C++ notamment en raison des contraintes matérielles (gestion des entrées/sorties) et des bibliothèques associées.

La carte Arduino peut accepter des *shields*, qui sont des extensions matérielles à entrer dans les ports appropriés et qui permettent d'étendre ses capacités, par exemple un *shield* Ethernet ou un *shield* Zigbee pour pouvoir utiliser les technologies associées.

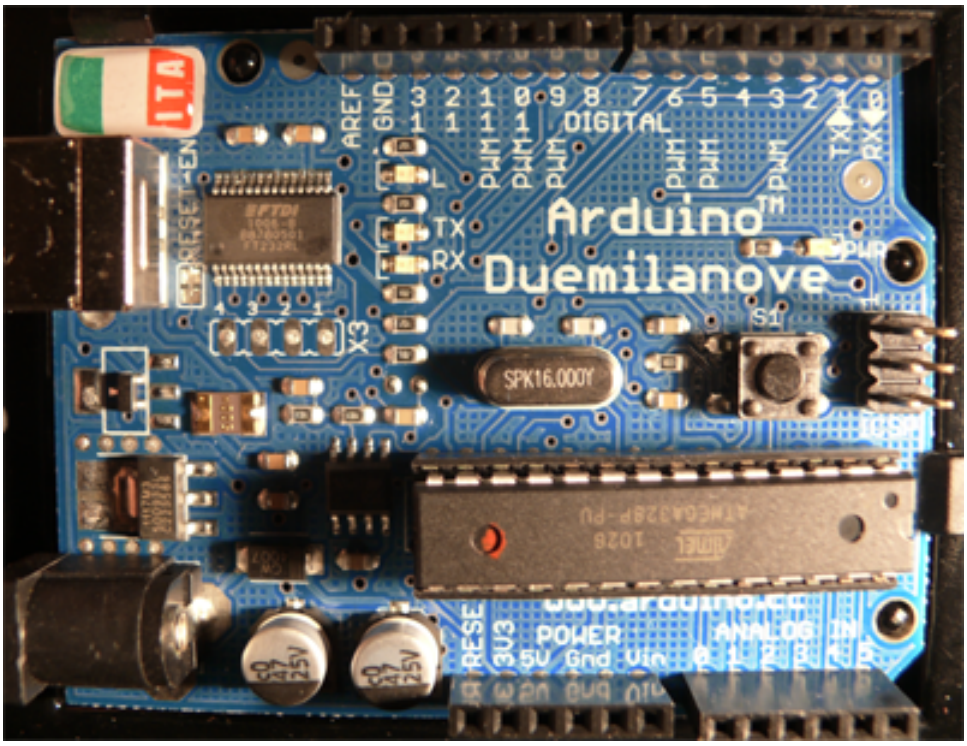

FIGURE 1 – Photo de la carte Arduino Duemilanove

4.1 Communication

Les cartes Arduino *Uno* et *Duemilanove* possèdent entre autre un micro-contrôleur ATMega328. L'environnement d'Arduino possède un terminal simplifié permettant un dialogue via le port série. Le dialogue permet l'envoi et la réception de données textuelles entre le PC et la carte Arduino. C'est grâce à ce port série que l'on communique avec les différents modules disponibles (GSM, Xbee, ...), avec des primitives minimales (`Serial.print()`, `Serial.read()`, ...).

4.2 Alimentation

Les cartes peuvent être alimentées soit par USB, soit par une alimentation externe. Bien qu'elles peuvent admettre une alimentation externe allant de 6 à 20V, l'alimentation recommandée et optimale doit être comprise entre 7 et 12V.

4.3 Mémoire

Le chipset ATmega328 possède 32 KB (donc 0.5 KB pour le bootloader) de mémoire pour le code compilé et embarqué, 2 KB de SDRAM et 1 KB de mémoire EEPROM[1].

4.4 Entrées-Sorties

Chacun des 14 pins numériques peut être utilisé comme une entrée ou une sortie à l'aide des fonctions `pinMode()`, `digitalWrite()`, et `digitalRead()`.

1. Electrically-erasable programmable read-only memory : Un type de mémoire morte

Certains pins ont des fonctions spéciales :
- Port série : 0 (RX) et 1 (TX). Utilisés pour recevoir (RX) et transmettre (TX) des données directement à travers le port série
- Interrupteurs externes : 2 et 3. Ces pins peuvent être configurés pour déclencher une interruption sur une faible valeur, un front montant/descendant ou un changement de valeur
- Les pins : 3, 5, 6, 9, 10, et 11 fournissent une sortie 8-bit
- Les pins : 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK) supportent une communication SPI[2]

Les cartes ont 6 entrées analogiques, marquées A0 à A5. Chacune d'elle fournit 10 bits de résolution (i.e. 1024 valeurs différentes). Par défaut, leur champs de mesure va de 0V à 5V (valeurs modifiables). Elles possèdent aussi un bouton `Reset` utilisé pour réinitialiser le chipset et redémarrer le programme embarqué.

Toute la documentation est disponible sur le site officiel d'Arduino pour la partie matérielle[14] et logicielle[15].

5 Environnement de développement Arduino

Arduino possède sa propre interface de développement que l'on peut télécharger sur le site officiel[11]. La démarche d'installation n'est bien évidemment pas la même en fonction des OS et/ou versions utilisées.

5.1 Linux Ubuntu

Version 10.10

Ubuntu 10.10 possède déjà la suite Arduino dans ses dépôts officiels. L'installation se fait donc simplement :

```
$ : sudo apt-get install arduino
$ : arduino&
```

Version 10.04

Pour Ubuntu 10.04, il faut rajouter Arduino dans le dépôt manuellement :

```
$ : sudo add-apt-repository ppa:arduino-ubuntu-team
$ : sudo apt-get update
$ : sudo apt-get install arduino
$ : arduino&
```

Attention !

La version actuelle de l'EDI des dépôts est la v.18, et ne supporte pas la carte UNO. Il faut donc télécharger la dernière version (version 22) de l'EDI depuis le site officiel[12] en plus de l'installation du paquet, et utiliser le binaire de l'archive à la place de celui du dépôt. De plus, le paquet "`brltty`[3]" semble (d'après le site officiel) rentrer en conflit avec les paquets Arduino.

2. Serial Peripheral Interface : un bus série synchrone, maître/esclave, full duplex de communication entre le micro-contrôleur et ses périphériques

3. BRLTTY est un démon qui fournit à Linux l'accès en mode texte au terminal pour une personne aveugle utilisant un affichage en braille

Il convient donc de le désinstaller :

```
$ : sudo apt-get remove brltty
```

5.2 Windows Vista & Seven

Il faut télécharger l'archive directement sur le site officiel[13], la dézipper, puis lancer "arduino.exe" pour lancer l'EDI. L'installation du périphérique USB est automatique pour la carte DUEMILANOVE dès le branchement. Il faut par contre mettre à jour le driver de la carte UNO, depuis le gestionnaire de périphériques (le fichier driver *ArduinoUNO.inf* se trouvant dans le dossier *Drivers* de l'archive).

Liens qui nous ont été utiles :
- http://www.arduino.cc/playground/Learning/Linux
- http://arduino.cc/en/Guide/Windows

6 Environnement de développement Android

Android est un système d'exploitation pour terminaux mobiles. Son environnement de développement permet de déployer des applications adaptées à ses spécificités, écrites dans le langage de programmation Java. Pour écrire ce code Java, nous avons utilisés l'environnement de développement intégré Eclipse.

6.1 Manuel d'installation et de configuration

Installation Linux & Windows

Cette partie est commune aux deux systèmes d'exploitations, à la différence qu'elle n'a pas besoin d'être faite sur les versions les plus récentes d'Ubuntu, où toutes les étapes ont déjà été effectuées. Avant toute chose, il faut donc vérifier la présence (et dans le cas échéant les installer dans l'ordre indiqué) de :

- Java Standard Edition v6 Development Kit (Java SE 6 JDK)[9]
- Android Software Development Kit (Android SDK)[3]
- Eclipse[7]

Il faut ensuite mettre à jour Eclipse avec la suite ADT, les Android Development Tools[1]. Pour ce faire, une fois Eclipse lancé, il faut faire *Help* > *Install New Software...* > *Add*, et y entrer l'URL d'ADT[2]. Une fois ADT téléchargé et installé, Eclipse doit être redémarré.

Configuration

A partir de cette étape, même sur les versions les plus récentes d'Ubuntu, il est obligatoire de procéder aux opérations suivantes :

- Configurer Eclipse dans *Windows* > *Preferences* > *Android* > *SDK Location* : indiquer le lieu l'installation du SDK Android

- Mettre à jour ADT dans *Windows* > *Android SDK and AVD Manager* > *Available packages* > *Android Repository* : indiquer les plate-formes Android dont l'on souhaite disposer. Il est nécessaire d'en avoir au moins une, et les versions exactes requises dépendent des "cibles" que vous souhaitez atteindre (ex : un téléphone en 1.6, etc.)
- Créer un AVD dans *Windows* > *Android SDK and AVD Manager* > *Virtual devices* > *New...* : indiquer dans *Target* la plate-forme Android pour laquelle on pourra l'utiliser (ex : Android 2.3.3 - API Level 10) et dans *SD Card* mettre une taille (ex - Size : 9 MiB)

Un AVD est un Android Virtual Device. Il sert à émuler une plate-forme Android pour tester les projets.

6.2 Gestion d'un projet Android

Création

Dans l'affichage des projets d'Eclipse, il faut faire clic-droit > *New* > *Project* > *Android Project*, et en remplir les spécifications. En voila un exemple, pour un projet HelloAndroid :

```
* Project name: HelloAndroid
* Application name: Hello, Android
* Package name: com.example.helloandroid (ou n'importe quel nom tant qu'il est de la forme com.xxx)
* Create Activity: HelloAndroid
* Min SDK Version: 10 (dépend de la plate-forme Android installée et de celle de l'AVD).
```

Il faut ensuite remplir ce projet de code. Encore une fois, voila un exemple qui affiche "Hello, Android" à l'écran :

```
package com.android.helloandroid;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        tv.setText("Hello, Android");
        setContentView(tv);
    }
}
```

Lancer l'option Run sur ce code va lancer l'AVD, l'émulateur de terminal Android. Attention : les AVDs viennent parfois avec des programmes pré-installés, comme une démonstration de l'API, qui peuvent alors entrer en conflit avec les votres. Il faut dans ce cas les désinstaller.

Connexion avec un terminal

Cette partie n'est nécessaire que pour les tablettes Android, et en particulier a été écrit pour les tablettes Android Ideos S7 nous ayant été fournies, bien que la démarche soit similaire pour d'autres variantes.

Une fois la tablette reliée à l'ordinateur par son cordon USB, il faut, dans le menu de celle-ci, choisir les Applications (le logo avec quatre carrés) et aller dans *Settings* > *Applications* > *Development* pour y allumer le USB Debugging. Attention, la tablette va émettre un bruit assez fort, pensez à la mettre en mode Silencieux avant !

Une fois cette étape faite, il faut que votre ordinateur puisse reconnaitre la tablette. La démarche est différente si vous êtes sur Linux[8] ou sur Windows (il faut télécharger le driver sur le site du constructeur[6]).

A partir de maintenant, et quand la tablette est connectée, dans Eclipse et son panneau *Run* > *Run* la tablette Ideos sera disponible. Il faut la sélectionner pour lancer le programme dessus plutôt que sur un AVD. L'application-test devrait alors normalement se lancer sur la tablette.

Attention !

Il faut d'abord vérifier la version d'Android installée sur la tablette, et éventuellement télécharger l'API appropriée pour ADT et s'assurer que le projet y corresponde (les tablettes fournies sont en version 2.1, voir **Configuration** plus haut). Il est alors nécessaire de changer le Project Build Target (alias le niveau de l'API et de la plate-forme) dans les Propriétés du projet. Il est parfois nécessaire que le projet soit flaggé comme debuggable dans le fichier *AndroidManifest.xml* en ajoutant *android :debuggable="true"* à la balise <application> pour que le transfert fonctionne.

Il est également possible que l'importation d'un projet déjà existant sous Eclipse provoque des erreurs. Dans ce cas, il faut le sélectionner, et faire clic-droit > *Android Tools* > *Fix Project Properties* puis le rafraichir (clic-droit > *Refresh*), voir le nettoyer (*Project* > *Clean*).

6.3 Programmation Android avancée

L'API Google

Google offre une API standard permettant de "piocher" des fonctionnalités dans certains de leurs services, comme Google Map.

- Dans Eclipse, faire *Windows* > *Android SDK and AVD Manager* > *Available packages*, et y sélectionner *Third-Party Addons*. Télécharger les API Googles correspondant à celles de votre SDK (ex : Android SDK version 2.1 correspond à l'API de niveau 7, prenez donc la Google API 2.1 correspondant au niveau 7)
- Clic-droit sur le projet, *Properties* > *Android* : sélectionner l'API Google correspondant à celui du projet
- Vous pouvez maintenant utiliser des applications Google telles que Google Map, fournies dans le package com.google.android
- Attention, vous devez créer un nouvel AVD basé sur cette API si vous voulez pouvoir l'utiliser en émulateur

La programmation Android en utilisant l'API Google nécessite souvent des permissions et bibliothèques sup-

plémentaires qu'il faut signaler dans le fichier *AndroidManifest.xml*. De même, pour certaines fonctionnalités telles que Google Map, il faut s'enregistrer aux services Google[5] par rapport à l'empreinte de votre ordinateur.

Utiliser deux AVDs

Particulièrement utile dans le cas, par exemple, d'applications basées sur l'envoi et la réception de SMS, il est possible de créer deux AVDs (prendre garde à leur donner le même niveau et les mêmes paramètres) et de les lancer en même temps en faisant Run deux fois et en sélectionnant un AVD différent à chaque fois. Les AVDs peuvent alors communiquer entre eux ! Par exemple, le numéro s'affichant en haut à gauche de la fenêtre de l'AVD (ex : 5554 :<nom AVD1>, 5556 :<nom AVD2>) est celui qu'il faut entrer pour les tests d'appels téléphoniques et d'envoi/réception de SMS. Remplacez juste le 0556XXXXXX habituel par 5554 ou 5556.

Liens utiles

- http://developer.android.com/sdk/installing.html
- http://developer.android.com/sdk/eclipse-adt.html
- http://developer.android.com/guide/developing/eclipse-adt.html
- http://developer.android.com/guide/tutorials/hello-world.html
- http://developer.android.com/resources/tutorials/views/hello-mapview.html

II Présentation des *shields* Arduino

1 *Shield* XBee

1.1 Fonctionnement

Le *shield* Xbee permet à une carte Arduino de communiquer sans fil en utilisant la technologie Zigbee. Le module peut communiquer jusqu'à 30 mètres à l'intérieur et 90 mètres à l'extérieur (sur zone dégagée). Il suffit, pour faire communiquer deux modules, d'écrire sur le port série. La chaîne de caractère est alors directement broadcastée, et les modules récepteurs affichent alors la chaîne reçue sur le port série.

Formatage des données au sein du port série

Le port série autorise le formatage des données sur son flux d'écriture/lecture. Ainsi, il est possible d'écrire au format `byte` (format brut), `bin` (binaire), `oct` (octal), `hex` (hexadécimal), `dec` (décimal), ainsi que flottant.

Exemples :

- Serial.print(78, BYTE) donne "N"
- Serial.print(78, BIN) donne "1001110"
- Serial.print(78, OCT) donne "116"
- Serial.print(78, DEC) donne "78"
- Serial.print(78, HEX) donne "4E"
- Serial.println(1.23456, 0) donne "1"
- Serial.println(1.23456, 2) donne "1.23"
- Serial.println(1.23456, 4) donne "1.2346"

Il faut donc connaître du côté réception le format des données pour éviter leur corruption.

1.2 Exemple d'application - Clignotement de LED à distance

Cet exemple présente le dialogue entre deux modules Xbee, l'un donnant l'ordre de faire clignoter une LED branchée sur l'autre module.

Le module récepteur a donc une LED de branchée sur un pin déclaré en `OUTPUT`. Il effectue une attente active en lecture sur le port série. Dès qu'un entier est lu (1 dans notre cas) il allume la LED, sinon il l'éteint.

```
int incomingByte = 0;
int led = 8; //Declare la led branchee sur la pin 8

void setup(){
  Serial.begin(9600); // Demarre le port serie a une vitesse de 9600 bps.
  pinMode(led, OUTPUT);
}

void loop(){
  if (Serial.available() > 0) {
    incomingByte = Serial.read();
  }
  if(incomingByte == '0'){
    digitalWrite(led, LOW);
  }
  else if (incomingByte == '1'){
    digitalWrite(led, HIGH);
  }
}
```

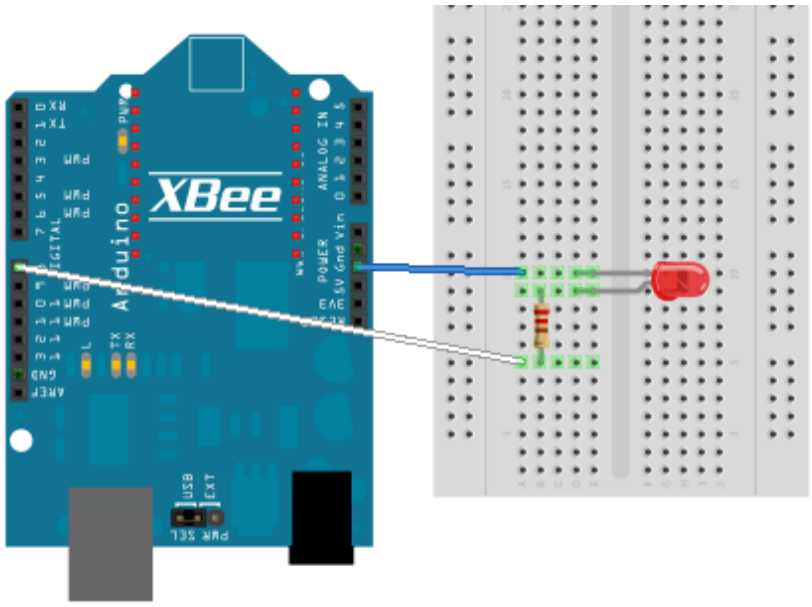

FIGURE 2 – Code & branchement du récepteur

Le module émetteur écrit simplement des suites de 0 et de 1 (selon une fréquence choisie) sur le port série. Le *shield* Xbee se charge lui d'envoyer l'information sur les ondes.

```
void setup(){
  Serial.begin(9600);
}

void loop(){
  Serial.print(1);
  delay(500);
  Serial.print(0);
  delay(1000);
}
```

FIGURE 3 – Code de l'émetteur

1.3 Problèmes rencontrés

Nous avons eu beaucoup de problèmes pour réussir à envoyer du code sur la board lorsque le module Xbee était branché. En fait, nous nous sommes rendus compte qu'envoyer du code avec le module Xbee branché n'était pas possible, car le port série communiquait alors avec le chipset Xbee et non avec le chipset Arduino.

Ainsi, nous sommes obligés, à chaque envoi de code, de débrancher le module Xbee puis de le rebrancher une fois le code uploadé, et ce sur chaque mainboard. Cela est donc assez contraignant au niveau du temps et des manipulations à réaliser.

2 *Shield* Wi-Fi

2.1 Fonctionnement

L'utilisation et la configuration du module Wi-Fi se fait par le port série grâce à des commandes du type `set wlan auth <value>`, `join <ssid>`, `open <addr> <port>` etc...

Cependant, une bibliothèque[4] expérimentale existe pour ce module, rendant la chose beaucoup plus simple. Nous y avons juste ajoutés une fonction *enableDHCP()* qui active l'acquisition automatique d'adresses IP par le protocole DHCP. Il faut laisser au module le temps de se connecter au réseau, d'acquérir une IP, d'effectuer les requêtes etc... afin d'éviter tout problème de synchronisation entre le code et la carte Wi-Fi. Trois diodes sont présentes sur la carte Wi-Fi pour nous informer de l'état de la connexion, suivant le fonctionnement suivant [4] :

Condition	Red LED	Yellow LED	Green LED
On solid	Not associated		Connected over TCP
Fast blink		Rx/Tx data transfer	No IP address
Slow blink	Associated, No Internet,		IP address OK
OFF	Associated, Internet OK		

2.2 Exemple d'application - Hébergement d'un serveur web

Les modules Wi-Fi peuvent *simuler* un serveur web en envoyant sur une socket (ouverte sur le port 80) des chaînes de caractères formatées en requêtes HTTP. Le programme consiste alors en une simple acquisition d'adresse

4. Tableau extrait du *Reference Guide*[10]

IP (en mode infrastructure sur cet exemple), une attente de connexion entrante sur la socket (port 80) et l'envoi d'une chaine HTTP répondant à un GET.

```
#include "WiFly.h"

Server server(80);

void setup() {
  WiFly.begin();
  [...]
}

void loop() {
  Client client = server.available();
  if (client) {
  boolean current_line_is_blank = true;
  while (client.connected()) {
    if (client.available()) {
      char c = client.read();
      if (c == '\n' && current_line_is_blank) {
        client.println("HTTP/1.1 200 OK");
        client.println("Content-Type: text/html");
        client.println();
        client.println("Hello world !");
        break;
      }
      if (c == '\n') {
        current_line_is_blank = true;
      }
      else if (c != '\r') {
        current_line_is_blank = false;
      }
    }
  }
  delay(100);
  client.stop();
}
```

La connexion s'effectue simplement en ouvrant un navigateur sur l'url $http : \backslash\backslash @IP_MODULE$.

2.3 Problèmes rencontrés

Utilisation d'un réseau en mode infrastructure

Le premier problème auquel nous nous sommes heurtés a été la non possession d'un réseau Wi-Fi infrastructure pour pouvoir s'y connecter et effectuer nos tests. En effet, les réseaux Wi-Fi disponibles sur le campus sont des réseaux avec des protocoles de connexion sécurisés, inutilisables avec nos modules Arduino.

Nous avons résolu le problème en utilisant un routeur propageant un réseau Wi-Fi en mode infrastructure.

***Shield* d'origine inutilisable**

En effet, lors de sa livraison, le *shield* Wi-Fi n'était pas utilisable en l'état actuel puisqu'aucun moyen de brancher le *shield* sur la carte Arduino n'existait. Nous avons donc soudé directement sur le *shield* des broches pour pouvoir à la fois brancher le *shield* sur la carte Arduino, et à la fois brancher d'autres entrées sur le *shield* Wi-Fi.

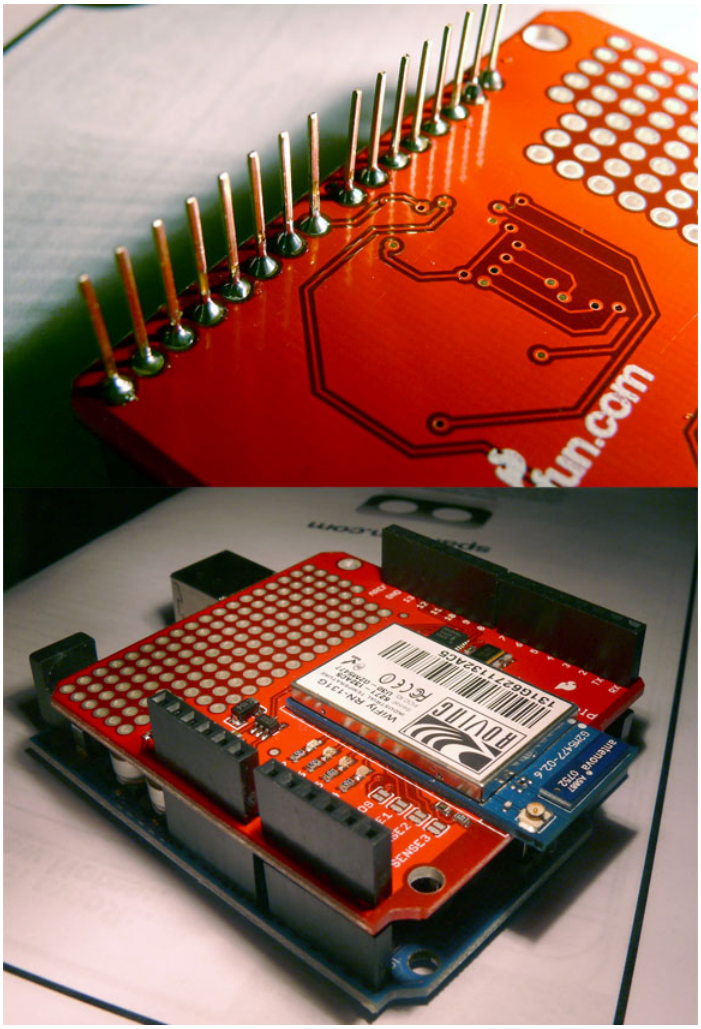

FIGURE 4 – Résultat de la soudure du *shield* Wi-Fi

Extinction du *shield* après un temps d'inactivité

Il se trouve qu'après un certain moment d'inactivité (environ 30 secondes), le module Wi-Fi n'est plus joignable (ni par socket ni par ping). Après de longues recherches pour résoudre le problème (notamment les temps d'inactivité par défaut), nous avons utilisés le programme *HardwareFactoryReset* de la bibliothèque Wifly[4] citée plus haut, remettant les modules WiFly dans leurs paramètres d'usine. Ils marchent maintenant normalement, sans s'éteindre.

3 *Shield* GPS

3.1 Fonctionnement

Il existe une multitude de *shield* GPS compatibles avec Arduino. Chacun a un mode de fonctionnement différent. Le *shield* qui nous a été fourni est le DS-GPM.S. Ce module GPS possède une connexion de type I^2C qui permet de communiquer avec seulement deux entrées analogiques (entrées 4 et 5), laissant ainsi le port série libre pour les autres modules (comme le GSM ou le Wi-Fi). La seule opération que l'on puisse faire sur le module GPS est la lecture des valeurs des différents registres (112 registres au total).

En outre, le module GPS possède 4 jumpers qui permettent de configurer l'adresse de lecture, ce qui fait qu'il est possible de brancher 4 modules en même temps qui utilisent tous la communication I^2C, et ceci en jouant sur la valeur de l'adresse de lecture.

3.2 Exemple d'application - Affichage des coordonnées sur le port série

Ce code lit les différentes valeurs de la latitude et longitude directement dans les registres du module GPS, puis écrit les coordonnées formatées sur le port série. La fonction `GetSingle()` sert à retourner l'octet contenu à l'adresse *Address*.

```
#include <Wire.h>
#define GPM 0x68

byte Address;
byte Data;
void setup()
{
  Wire.begin();
  Serial.begin(9600);
  get_infos();

}
void get_infos(){

  Serial.print("Latitude: ");
  Address = 14;
  Data = GetSingle();
  Serial.print(Data, DEC);
  [...]
  Address = 32;
  Data = GetSingle();
  Serial.println(Data, BYTE);
}
```

3.3 Problèmes rencontrés

Comme le *shield* Wi-Fi, le *shield* GPS nécessite des soudures pour être connecté à la carte Arduino.

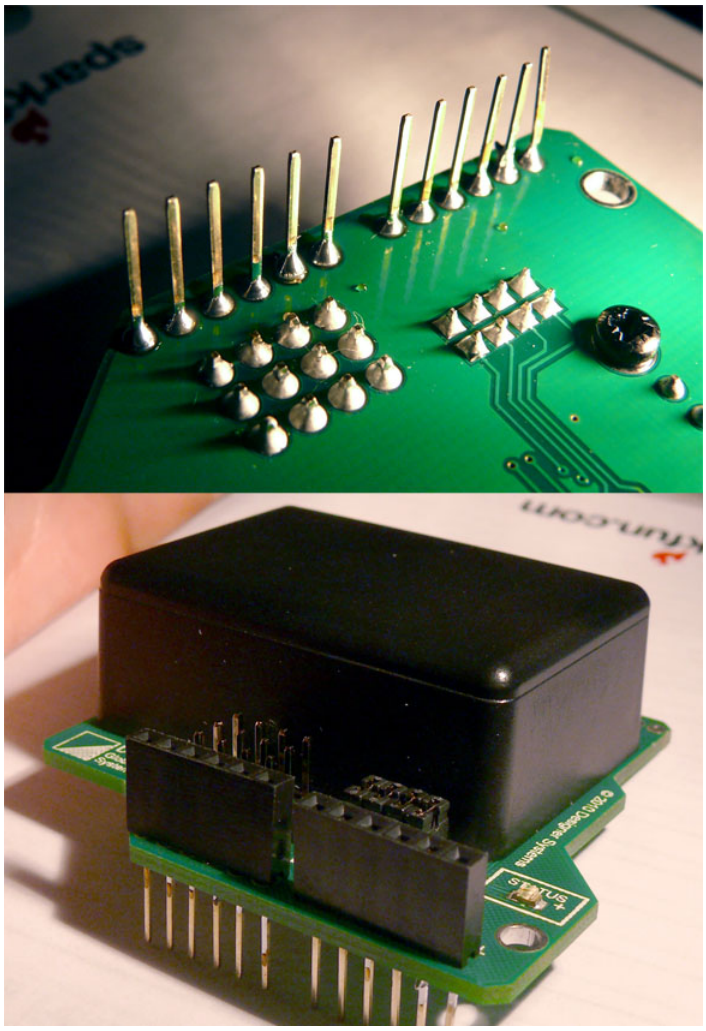

FIGURE 5 – Résultat de la soudure du *shield* GPS

Lorsque le module est déplacé sur une grande distance en étant éteint, son temps de synchronisation avec les satellites peut être relativement long (jusqu'à 30min). Une LED bleue sur le module indique l'état de synchronisation du GPS :

- Fixe : le module n'est pas synchronisé.
- Clignotement lent : le module est synchronisé et immobile.
- Clignotement rapide : le module est synchronisé et mobile.

Il est également possible de connaître l'état du module en lisant le quatrième et le cinquième bit du registre 112.

4 *Shield* GSM/GPRS

4.1 Fonctionnement

Le dialogue avec le module GSM via le terminal du port série n'a posé aucun problème, et nous avons pu *communiquer* avec le module sans souci particulier.

En effet, comme le module Xbee, les fonctionnalités du module GSM sont accessibles via le port série grâce à des commandes `AT`.

Principales commandes que nous avons utilisé sur le module GSM :

Commandes	Description
AT	État de la connexion au port série
AT+CPIN="****"	Débloquage du PIN
AT+COPS ?	Retourne le nom du réseau du mobile
ATD**********;	Effectue un appel vocal
ATH	Termine un appel en cours
AT+CSCA ?	Récupère le numéro du centre d'appel SMS
AT+CMGF=1	Sélectionne le format pour les SMS (1 pour format texte)
AT+CMGS="**********"	Spécifie le numéro du destinataire pour les SMS
AT+CNMI=2,2,0,0	Permet la redirection de flux vers le port série

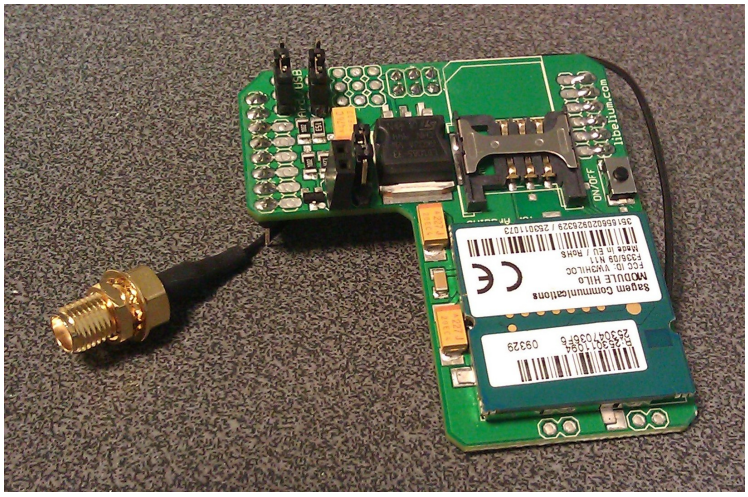

FIGURE 6 – *Shield* GSM

4.2 Exemple d'application - Envoi de SMS

```
int led = 13;
int onModulePin = 2;              // le PIN permettant d'allumer le module

void switchModule(){
  digitalWrite(onModulePin,HIGH);
  delay(2000);                    //Reboot du module
  digitalWrite(onModulePin,LOW);
}

void setup(){
  pinMode(led, OUTPUT);
  pinMode(onModulePin, OUTPUT);

  Serial.begin(19200);            // specifie la vitesse de transmission

  switchModule();                 // allume le module

 delay(5000);                     // attente de l'etablissement de la connexion


 Serial.println("AT+CMGF=1");     // selectionner le format texte pour
                                    les SMS
}

void loop(){
  Serial.print("AT+CMGS=");       // specifie le numero du destinataire
  Serial.print(34,BYTE);          // envoie du caractere "
  Serial.print("06********");
  Serial.println(34,BYTE);
  Serial.print("Hello World.");   // Texte a envoyer
  Serial.print(0x1A,BYTE);        // envoie du caractere hexa 1A qui marque
                                    la fin du message
  Serial.print("AT*PSCPOF");      // eteint le module GSM

}
```

4.3 Problèmes rencontrés

– Le premier problème rencontré était l'absence de l'antenne GSM qui empêchait toute opération d'envoi et de réception d'appel ou de SMS (message d'erreur *"NO CARRIER"*).

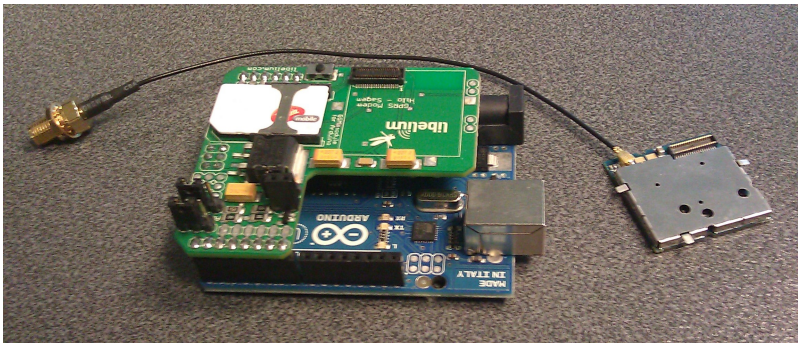

FIGURE 7 – *Shield* GSM - Branchement de l'adaptateur de l'antenne

– Une mauvaise configuration du taux de transmission engendrait un comportement du *shield* imprévisible et aléatoire. Un réglage à 19200 bps a permis de résoudre le problème.
– Au départ, des pertes de connexions récurrentes apparaissaient, or il est indiqué dans les spécifications du *shield* GSM qu'une simple alimentation par le port USB peut ne pas être suffisante pour alimenter le *shield* et ainsi empêcher d'obtenir une connexion stable. L'utilisation d'une alimentation externe (9V) a corrigé ce problème.
– Entre les différentes commandes envoyées au *shield*, il est parfois nécessaire d'utiliser des *delay()* pour permettre au *shield* d'avoir le temps de bien interpréter chaque commande.
– Un des plus gros problème rencontré a été le caractère hexadécimal de fin de message `0x1A` lors de l'envoi d'un SMS qui n'était pas interprété par l'environnement de développement Arduino. L'utilisation du logiciel *X-ctu* permettant l'interprétation de tels caractères a corrigé partiellement ce problème. Nous nous sommes finalement rendu compte que c'est la position des jumpers qui posait problème. Ils doivent être positionnés sur *Arduino* pour que cela fonctionne.

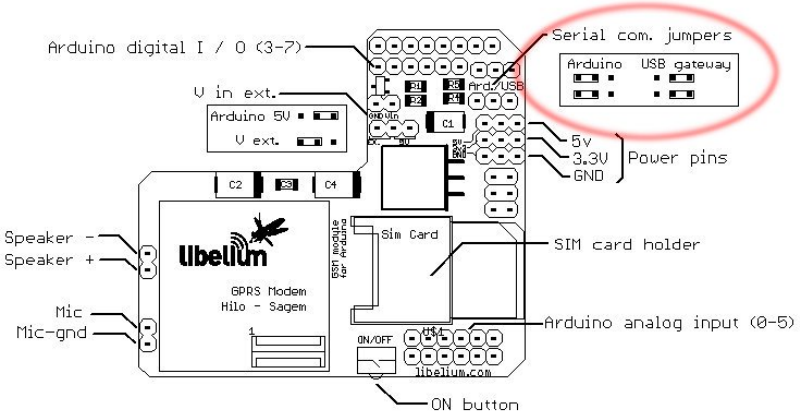

FIGURE 8 – Emplacement des jumpers sur la carte GSM[17]

III Élaboration du projet

1 Création & Configuration de réseaux Wi-Fi Arduino

Les *shields* Wi-Fi fournis permettent la création et la connexion de réseau Wi-Fi en mode infrastructure et ad hoc. Initialement, le projet devait se baser sur une communication dans ce dernier mode. L'écriture des programmes associés n'a pas posé de problèmes particuliers, avec en exemple le code pour la création d'un tel réseau :

```
void create_adhoc(char * ssid, int chanel) {
  Serial.begin(9600);
  SpiSerial.print("$$$"); //Passage en mode commande
  delay(800);
  SpiSerial.println("set wlan join 4"); //Configuration du mode Wi-Fi en adhoc
  delay(800);
  SpiSerial.print("set wlan ssid "); //Configuration du ssid
  SpiSerial.println(ssid);
  delay(800);
  SpiSerial.print("set wlan chan "); //Configuration du chanel
  SpiSerial.println(chanel);
  delay(800);
  SpiSerial.println("set ip dhcp 2"); //Activation du DHCP
  delay(800);
  SpiSerial.println("save"); //Sauvegarde de la configuration
  delay(800);
  SpiSerial.println("reboot"); //Rebooting...
  delay(3000);
  Serial.println("");
}
```

Les tests effectués ont permis de faire des échanges de données entre deux modules Arduino avec succès.

Problèmes rencontrés

Très rapidement, un problème majeur est apparu : il s'est révélé très difficile de connecter les tablettes Android que nous avions à un réseau Wi-Fi. En effet, le système d'exploitation Android ne gère nativement que les réseaux sans-fils en mode infrastructure, et nous n'avons finalement réussis à avancer qu'une fois en possession d'un routeur capable de connecter les modules Arduino et les tablettes.

Une solution alternative aurait pu être trouvée en partant sur la modification directe du système d'exploitation Android. Des tutoriaux sont disponibles sur Internet, mais cela aurait requis de remplacer des modules officiels par d'autres fournis par des services non-officiels et aurait de plus requis un piratage de la tablette pour nous attribuer les droits *root*. Nous avons donc préférés nous débrouiller d'une autre manière. De plus, si nous avions fait ça, nous n'aurions peut-être pas eu le temps de s'investir d'avantage sur les autres *shields*, de par la complexité de leurs problèmes individuels et le temps mis pour les régler.

2 Protocole de communication

Un protocole texte a été établi pour communiquer entre la tablette et les modules. Les requêtes envoyées par la tablette restent simples, puisqu'elles consistent à envoyer le caractère ♯ suivit par un chiffre, chaque chiffre correspondant à une commande particulière. L'utilisation de ce format est justifié car le caractère ♯ est rarement employé dans la vie courante (il n'y a donc pas de risques d'interférences), et surtout car la carte Arduino n'est capable de lire les flux entrant qu'octet par octet. L'utilisation de seulement 2 octets pour décrire une requête évite que le code ne devienne trop complexe et instable.

La réponse donnée par la carte Arduino est une chaine de caractères préfixée de 3 caractères indiquant la nature du module qui répond à la requête (*"GPS"* pour le module GPS, *"THM"* pour le module de température, etc...). Le reste de la chaine contient les données produites par le *shield* (par exemple les coordonnées GPS dans le cas d'une requête Arduino/GPS).

En fonction du moyen de communication utilisé (réseau Wi-Fi ou GSM), nous utilisons pour communiquer une socket TCP classique dans le premier cas, et un envoi de SMS standard dans le deuxième cas.

Interrogation des modules	
#1	Demande des coordonnées GPS
#2	Demande de la température
#3	Demande de l'état de la porte
#4	Demande de la luminosité
#?	Demande de la liste des services installés (Wi-Fi)

Préfixes des réponses des modules	
GPS	Réponse du GPS
THM	Réponse de la sonde de température
DOO	Réponse du la sonde de la porte
LUM	Réponse du la sonde de luminosité
OK	Liste les services (Wi-Fi uniquement)

3 Module tracker GPS

Le module de tracking GPS consiste en l'association d'une alimentation externe, du *shield* GPS, et d'un *shield* de communication (Wi-Fi ou GSM). Il permet, grâce à l'utilisation du protocole de communication sus-cité, de donner sa géolocalisation sur simple demande. Lors de la réception de la chaîne ♯1, le module met à jour ses coordonnées et les envoie à l'émetteur de la demande.

L'utilisation de l'alimentation externe permet une entière autonomie du module, et il peut être utilisé n'importe où - du moment qu'un accès au réseau GSM est possible, car le *shield* Wi-Fi aura rarement une portée suffisante pour être pertinent.

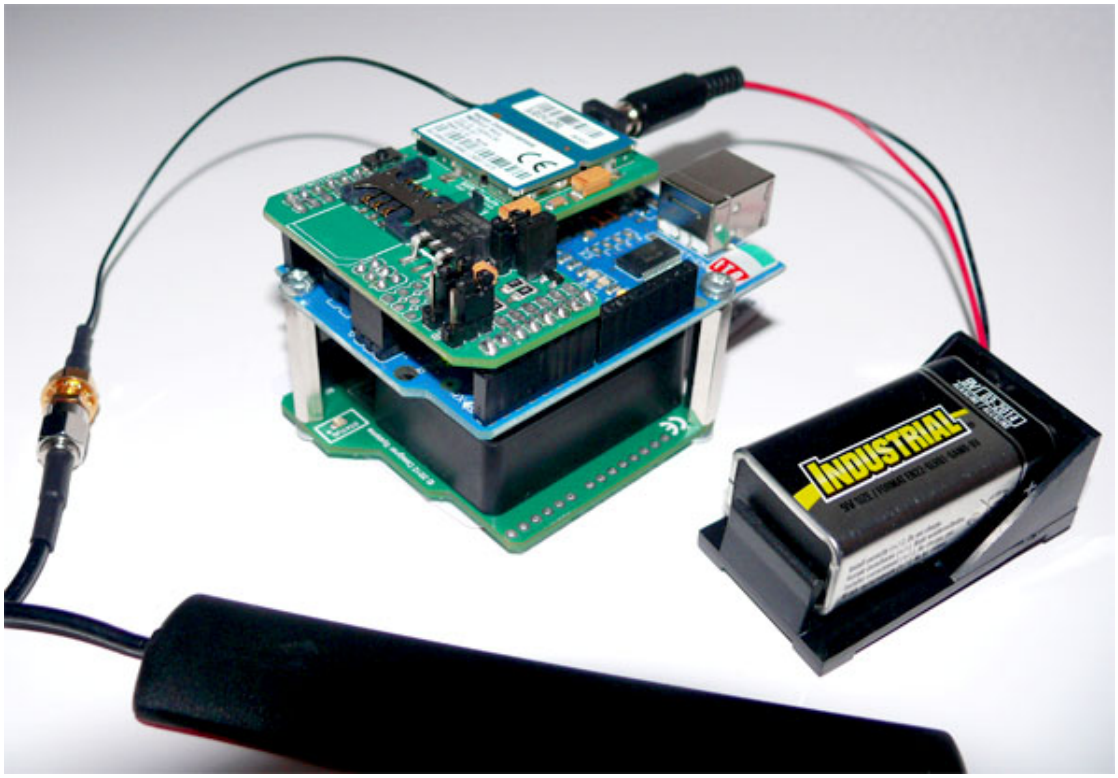

FIGURE 9 – Assemblage du module de tracker GPS

Le code suivant consiste à analyser les SMS entrants, puis à mettre à jour les coordonnées GPS pour les envoyer.

```
void loop(){
  if(Serial.available() > 0){
    parse_receiver(); //Permet de stocker le numero de l'emetteur du SMS entrant
    inchar = Serial.read();
    if (inchar == '#')
    {
      inchar = Serial.read();
      if (inchar=='1'){ //Si on recoit la chaine "#1"
        refresh_coor();
        send_sms_coor();
}}}}
```

4 Module d'information de luminosité

Ce module permet de renseigner la luminosité auquel il est exposé. Nous utilisons pour cela un *photoresistor* avec lequel nous calculons, grâce à sa résistance en sortie, le pourcentage de lumière associé.

Le code associé ouvre une socket en mode serveur, puis attends qu'un client s'y connecte et y envoie la suite de caractères ♯4. Lorsque cette chaîne est reçue, il écrit dans la socket l'information de luminosité après en avoir lu et

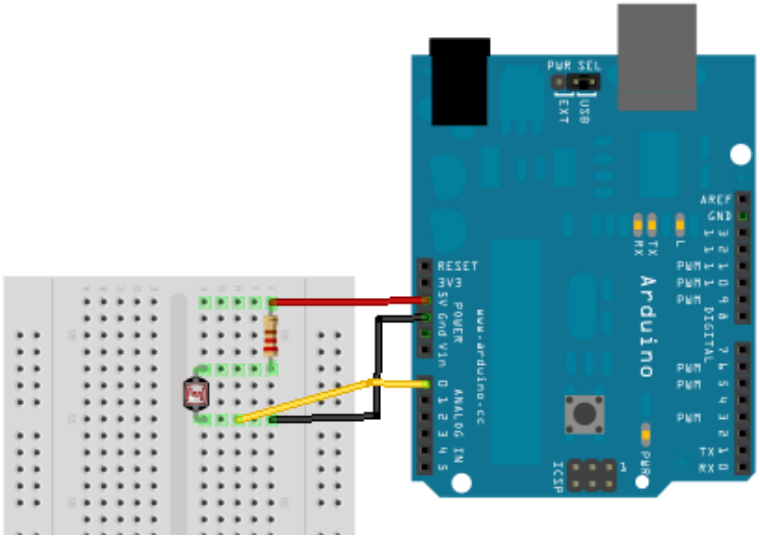

FIGURE 10 – Branchement du module de luminosité

calculé la valeur. Le code ci-dessous est un abrégé de la version Wi-Fi.

```
void loop() {
  Client client = server.available();
  if (client) {
    if (client.connected()) {
      while(client.available() < 1);
      char c1 = client.read();
      while(client.available() < 1);
      char c2 = client.read();
      if(c1=='#' && c2=='5'){
        getLumi(photo);
        client.println("Luminosite : ");
        client.println(lumi);
        client.println("%");
      }
      client.stop();
}}}
```

5 Module d'information de température

Ce module permet de renseigner la température ambiante en degrés Celsius. Nous utilisons pour cela une *thermorésistance* avec laquelle nous calculons la température associée.

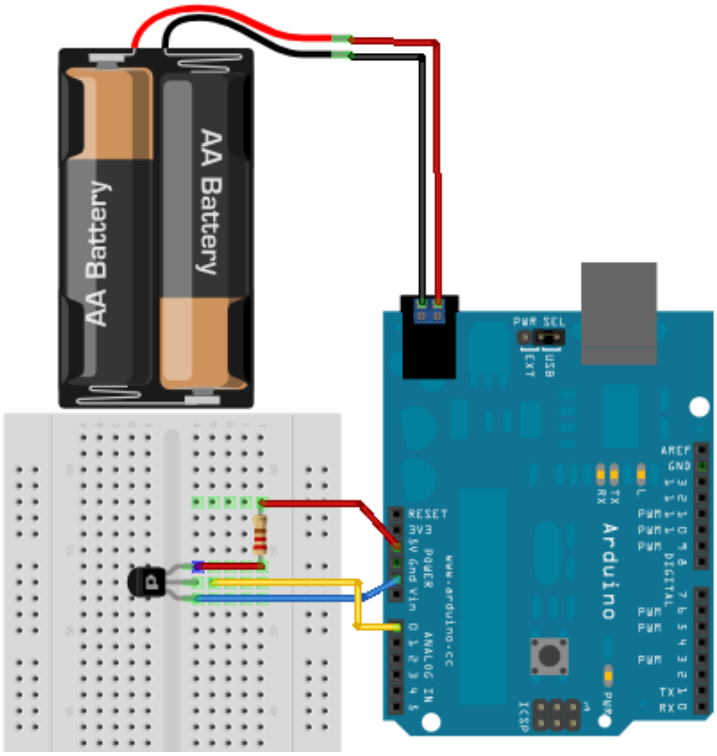

FIGURE 11 – Module d'information de température

Le code associé est très proche de celui du module de luminosité, à l'exception du calcul de la température :

```
[...]
client.println("Temp : ");
client.println((getVoltage(tmpin) - 0.5) * 100);
client.println("deg C");
[...]
```

6 Module de détection d'ouverture de porte

Ce module permet de renseigner l'état d'une porte (sur laquelle il est branché) pour savoir si celle-ci est ouverte ou fermée. Nous utilisons un *flex sensor* capable de détecter les torsions. En appliquant le *sensor* sur le coin de la porte (voir figure 12), l'ouverture de celle-ci va engendrer une torsion du module que l'on est capable de détecter avec la carte Arduino.

Le code associé ne diffère que très peu des autres modules, la seule différence consistant en la lecture de la valeur du *flex sensor*. Le code ci-dessous est celui de la version SMS.

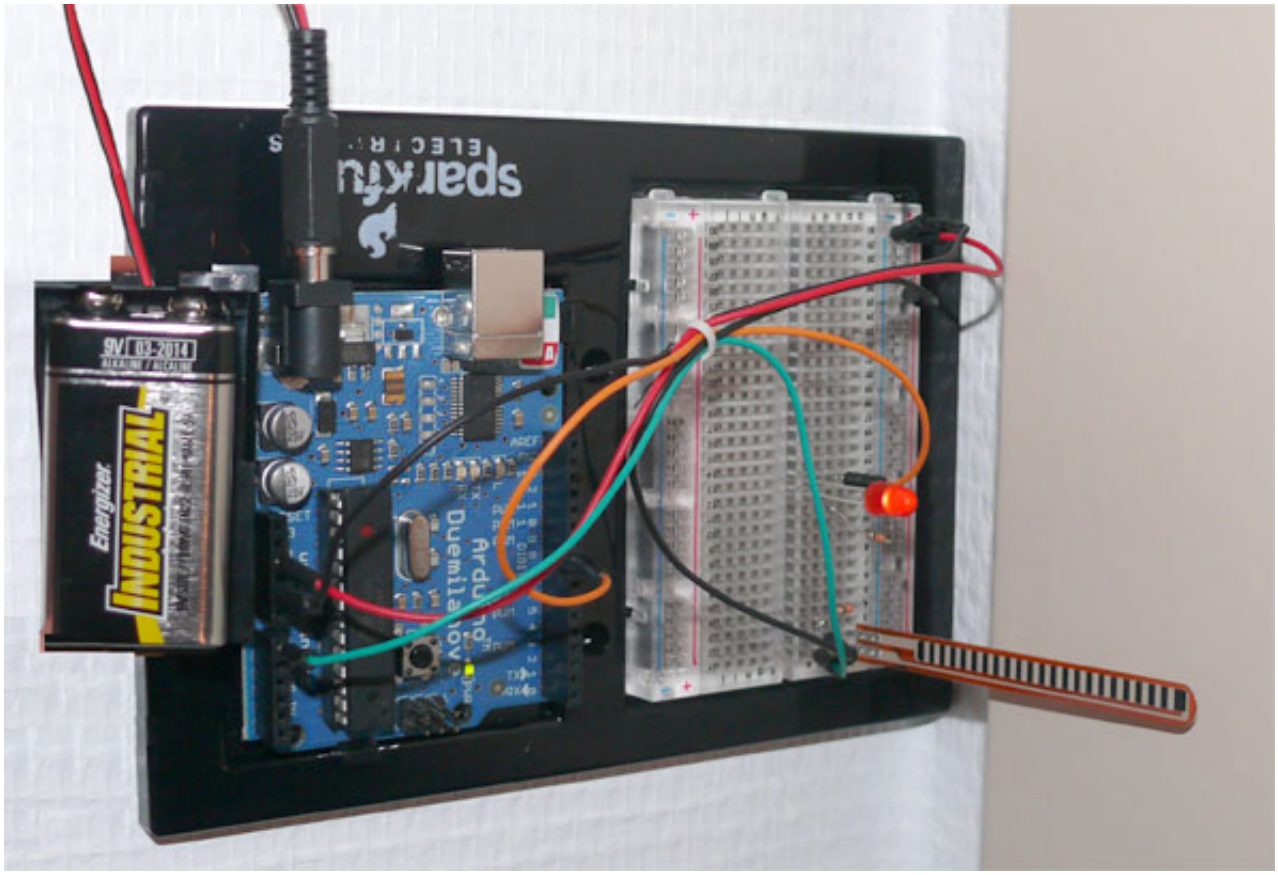

FIGURE 12 – Module du détecteur d'ouverture de porte

```
void loop(){
  if(Serial.available() > 0){
    parse_receiver();
    inchar = Serial.read();
    if (inchar == '#'){
      inchar = Serial.read();
      if (inchar=='3'){ //Si on recoit la chaine "#3" (exemple de commande)
        int val = analogRead(flex);
        if(val >= 276) send_sms(1); //276 etant la valeur mediane du
 flex sensor
        else send_sms(0);
      }
    }
  }
}
```

7 AWACS

7.1 Présentation

AWACS est le nom de la plate-forme d'administration d'équipements Arduino sur tablette Android que nous avons programmé. Son nom signifie **A**ndroid **W**ireless **A**dministration and **C**ontrol **S**ervice, et elle permet de communiquer avec les modules Arduino que nous avons construit tout au long de ce projet.

FIGURE 13 – Un AWACS est à l'origine un avion de guerre électronique[16]

A l'origine, cette plate-forme avait un but expérimental, et il était par exemple proposé de pouvoir implémenter une fonctionnalité "domotique" au sein du LaBRI, la tablette ayant alors pour but de communiquer avec des modules Arduino préalablement disposés dans le bâtiment. Par exemple, on aurait pu l'utiliser pour localiser les modules, les configurer à distance, et récupérer les informations qu'ils produisaient. La tablette Android ayant des capacités Wi-Fi et GSM, elle devait être capable d'interagir avec les modules par ces deux modes de communication.

7.2 Fonctionnalités

Actuellement, AWACS est capable d'envoyer et de recevoir des messages aussi bien par Wi-Fi que par SMS, puis de traiter ceux-ci afin de savoir s'ils correspondent au protocole d'échange que nous avons créés. Ce protocole est basé sur un système de tags, qui sont ensuite utilisés pour formater les messages échangés d'une manière particulière.

Ainsi, comme cela a déjà été décrit plus haut dans la section **Protocole de communication**, si AWACS reçoit un message contenant (exactement) la ligne "GPS 44 24.8191N 000 43.2812E Villeneuve-sur-Lot ArduinoUno", il comprendra aux trois premières lettres (le tag) qu'il va devoir envoyer les données à son module GPS. Les coordonnées qui suivent sont ensuite lues et transformées pour être acceptées par l'API Google Map accessible sur Android, tandis que les deux derniers mots sont des champs d'informations optionnels. Il est à noter que ce format d'écriture pour les coordonnées GPS est celui fourni par le *shield* GPS de nos modules Arduino.

Le module Wi-Fi d'AWACS permet de scanner la présence de modules Arduino dans la zone, de voir les *shields* dont ils sont équipés, et de leur demander des informations pertinentes. Par exemple, si un module Arduino équipé d'un module GPS et d'un module Thermomètre est détecté, on pourra lui demander sa géolocalisation ainsi que la température auquel il est exposé. AWACS dispose aussi d'un module pour envoyer des SMS, simple d'emploi.

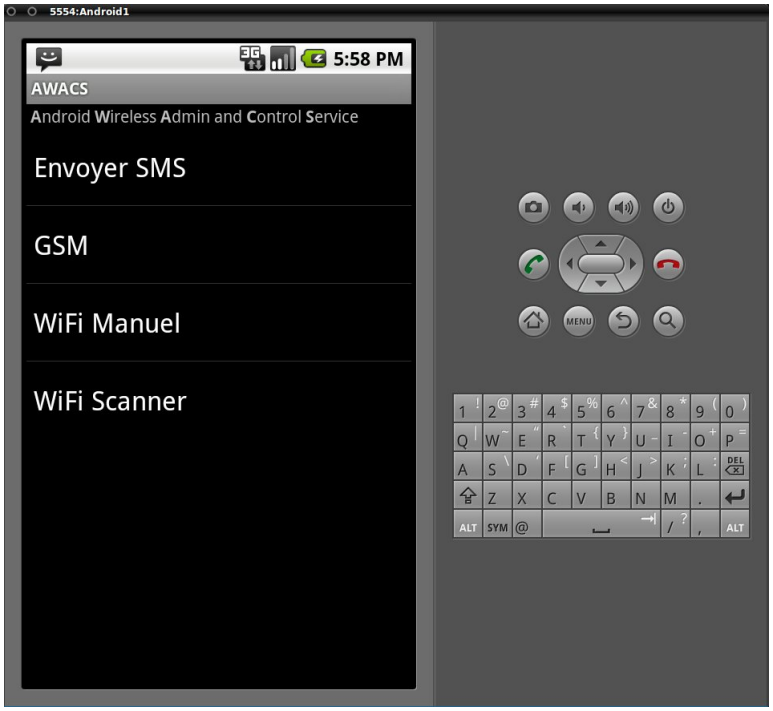

FIGURE 14 – Le menu d'AWACS

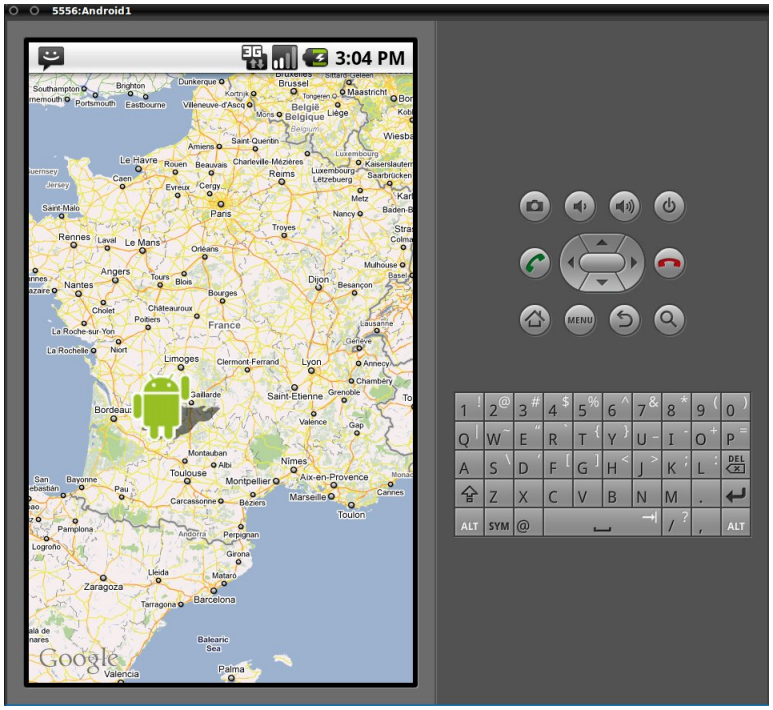

FIGURE 15 – AWACS qui communique avec un module GPS Arduino

Dans l'état actuel des choses, le seul retard d'AWACS par rapport à son cahier des charges est qu'il n'administre pas vraiment les modules Arduino, c'est à dire qu'il ne peut pas leur envoyer de messages pour les configurer à

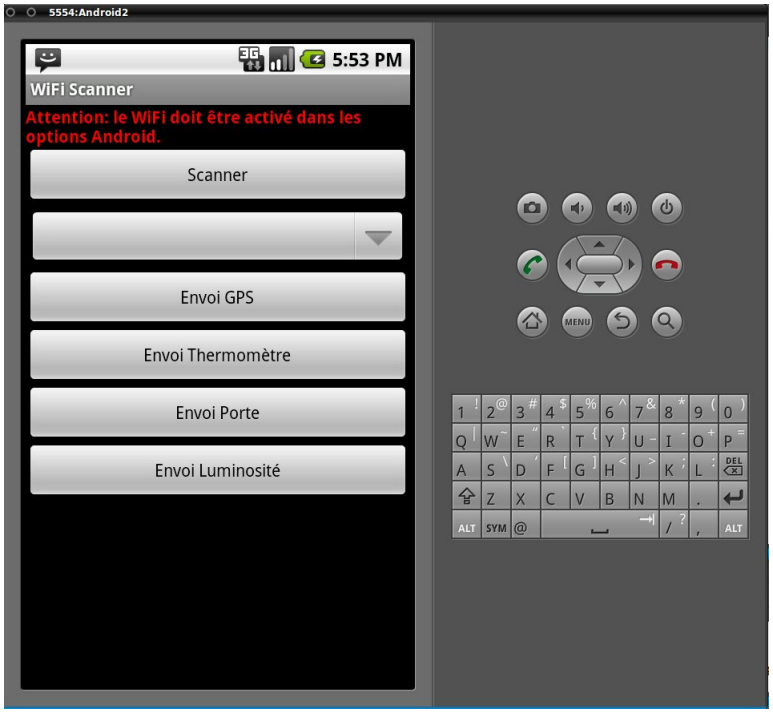

FIGURE 16 – Le module Wi-Fi

distance. Cependant, l'implémentation d'une telle administration aurait tout à fait été possible avec plus de temps pour le développement, et on aurait par exemple pu envisager de pouvoir re-configurer à distance le code PIN du *shield* GSM ou bien l'adressage IP du *shield* Wi-Fi.

L'architecture d'AWACS étant basée sur l'envoi et la réception de messages, il suffirait d'enrichir le protocole déjà mis en place avec de nouveaux tags pour pouvoir accepter de nouveaux modules, le système étant à chaque fois similaire (réception, envoi au bon module pour traitement, parsing du message pour extraire les informations). Les possibilités sont multiples, mais il est à noter que le matériel employé dispose de nombreuses limitations réduisant parfois ce que l'on peut en faire.

7.3 Problèmes rencontrés

Le développement d'AWACS a été ponctué de petits problèmes qui sont maintenant réglés. Ils ont tout de même fait perdre un temps certain, temps qui nous aurait permis de développer plus de fonctionnalités au programme.

Le problème majeur quant au développement d'AWACS a été le manque de matériel correct sur lequel travailler. En effet, en plus des problèmes de réseaux Wi-Fi mentionnés plus hauts, nous n'avions pas les droits nécessaires pour installer certains programmes sur nos comptes pourtant nécessaires au développement Arduino et Android, nous forçant à compter sur des solutions non-standardisées telles que nos ordinateurs personnels. Il faut toutefois noter que les administrateurs du CREMI nous ont aidés de leur mieux, et ainsi, par exemple, les ordinateurs du bâtiment reconnaissent maintenant de manière native les tablettes Android, ce qui n'était pas le cas avant le début de notre projet.

IV Conclusion

Le projet demandait que nous apprenions les spécificités du matériel Arduino et Android, que nous en rendions compte, et que nous développions une plate-forme capable de faire interagir les différents modules et programmes construits.

La principale difficulté rencontrée a clairement été le temps très limité de six semaines dont nous avons disposé, surtout au vu du fait que nous étions confrontés à de toutes nouvelles technologies et à un matériel électronique sur lequel nous n'avions aucune expérience. Toutefois, cela a pu être en partie compensé par le fait que la répartition du travail était aisée, chaque module étant indépendant et pouvant être travaillé en parallèle avec les autres.

Par contre, la découverte progressive des possibilités du matériel Arduino a empêché que le développement Android soit régulier, et il nous a souvent fallu changer de direction durant le développement. Cela explique le fait que la partie "administration" du programme Android soit manquante. Globalement, nous sommes satisfaits de l'avancement du projet, et même si nous ne remplissons pas toutes les clauses du cahier des charges original, les objectifs didactiques et techniques du projet ont pour nous été accomplis.

Le programme Android et les modules Arduino fonctionnent dans leur état actuel, même si avec plus de temps nous aurions pu ajouter plus de fonctionnalités et d'interopérabilité. Pour le futur, nul doute que le rapport ci-présent et les ressources fournies permettront à un novice de débuter beaucoup plus facilement que nous en développement Android et Arduino.

Bibliographie

[1] Android development tools - installation.
http://developer.android.com/sdk/eclipse-adt.html#installing.

[2] Android development tools - url eclipse.
https://dl-ssl.google.com/android/eclipse/.

[3] Android software development kit.
http://developer.android.com/sdk/index.html.

[4] Bibliothèque arduino pour le shield wifi.
https://github.com/sparkfun/WiFly-Shield.

[5] Demande de clé pour l'api google map.
http://code.google.com/android/add-ons/google-apis/mapkey.html.

[6] Driver de la tablette ideos - windows.
http://www.huaweidevice.com/worldwide/downloadCenter.do?method=list&flay=software&directoryId=20&treeId=0.

[7] Eclipse.
http://www.eclipse.org/downloads/.

[8] Installation de la tablette ideos - linux.
http://developer.android.com/guide/developing/device.html#setting-up.

[9] Java standard edition v6 development kit.
http://www.oracle.com/technetwork/java/javase/downloads/index.html.

[10] Rn-131g reference guide - wifly module.
http://www.sparkfun.com/datasheets/Wireless/WiFi/WiFlyGSX-um.pdf.

[11] Site officiel d'arduino.
http://www.arduino.cc.

[12] Site officiel d'arduino - lien vers l'archive arduino (version 22) pour linux.
http://arduino.googlecode.com/files/arduino-0022.tgz.

[13] Site officiel d'arduino - lien vers l'archive arduino (version 22) pour windows.
http://arduino.googlecode.com/files/arduino-0022.zip.

[14] Site officiel d'arduino - présentation de la carte uno.
http://arduino.cc/en/Main/ArduinoBoardUno.

[15] Site officiel d'arduino - référence logicielles.
http://arduino.cc/en/Reference/HomePage.

[16] Source de la photo de l'avion awacs : Us air force.
http://www.af.mil/shared/media/photodb/photos/021107-O-9999G-024.jpg.

[17] Source de la vue composants du shield gsm : Wiki libelium.
http://www.libelium.com/squidbee/index.php?title=New_GPRS_module_for_Arduino_(Hilo_-_Sagem).